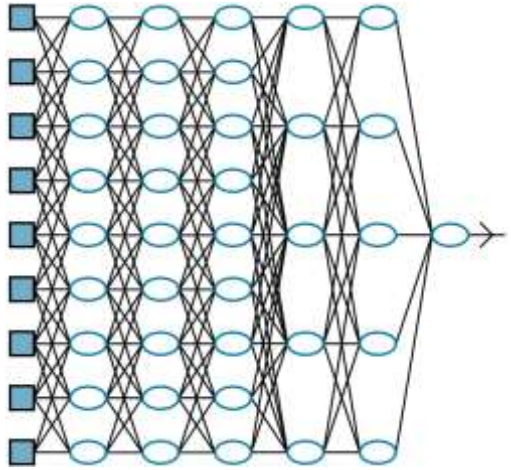
Neural Networks II

ITI0210, lecture 13 (2021)

Review

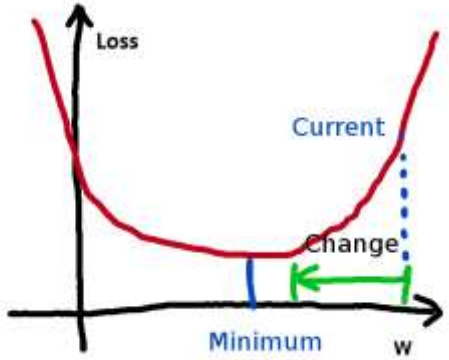
Neural networks: layers of units/artificial neurons



Weights control how information from inputs influences the output layer

Training: error on output is a function of weights

Change weights to reduce error



Training Networks

Common wisdom that also applies to neural nets

Training Screw-ups

- Not having enough/representative data
- No feedback to evaluate the model (how do I test?)
- Evaluating classifiers/predictors on training data

Example with these basics: face recognition

Maybe I should include other people?

What am I actually trying to do? And who will draw these boundary boxes?

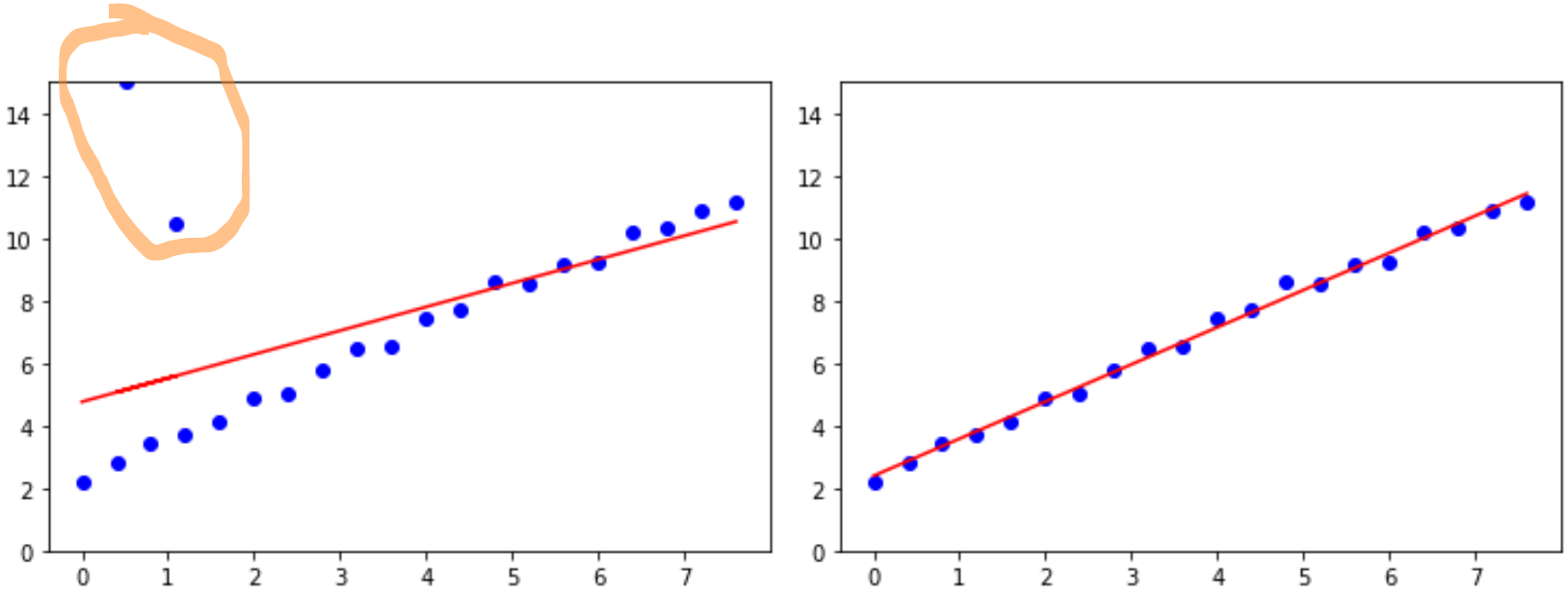
99.99% on training set

Awesome, does it work in my application now?

Images: Wikimedia Commons

Training Screw-ups

Noise: linear models fitted with and without a few noisy outliers



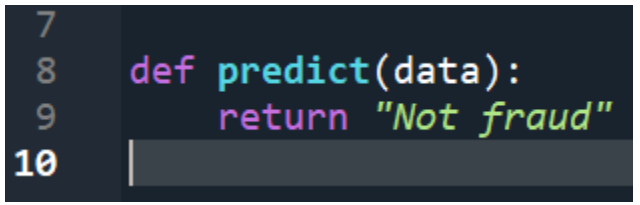
(Single neurons behave similarly to linear models)

Training Screw-ups

Unbalanced data:
10000000 card transactions, 1000 fraudulent

No joke: a model rewarded by (optimized by) accuracy adopts such strategies

99.99% accuracy predictor



```
7
8   def predict(data):
9       return "Not fraud"
10
```

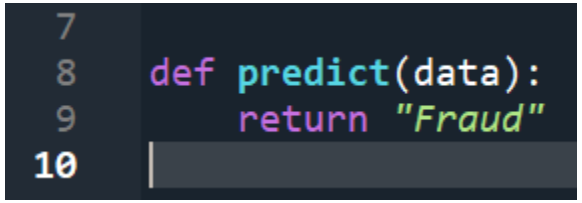
Fix: measure the right thing: $Recall = \frac{TP}{TP+FN}$

Constant "Not fraud" has $Recall = \frac{0}{0+1000} = 0$

TP – true positive (fraud)
FN – false negative (fraud)
$TP + FN$ – total fraud (detected and undetected)

~~Training~~ Evaluation Screw-ups

How about this amazing 100% recall predictor?
(unlikely to happen with neural networks)



```
7
8   def predict(data):
9       return "Fraud"
10
```

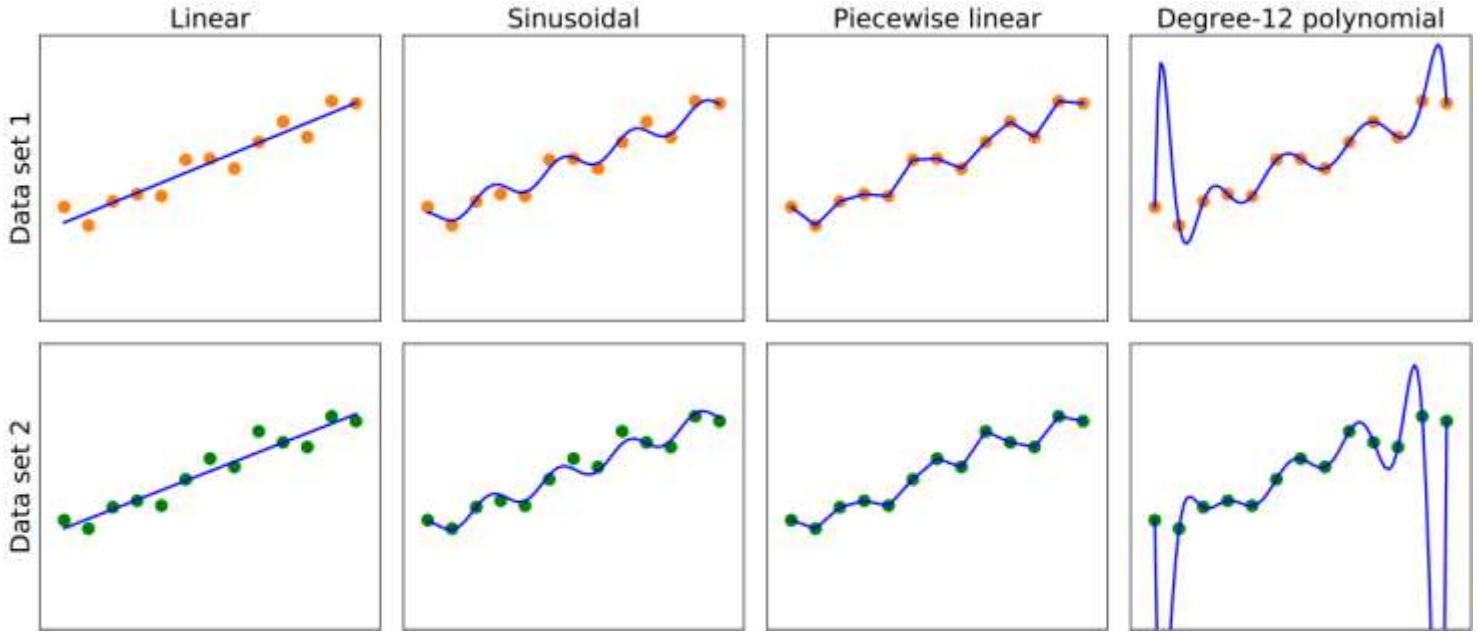
$Precision = \frac{TP}{TP+FP}$ metric to the rescue (FP – false positive)

Constant "Fraud" predictor: $Precision = \frac{1000}{10000000} = 0.01\%$

Conclusion: use the **right metric(s)** for **your use case**
(Proper training with unbalanced data is a separate issue)

Training Screw-ups

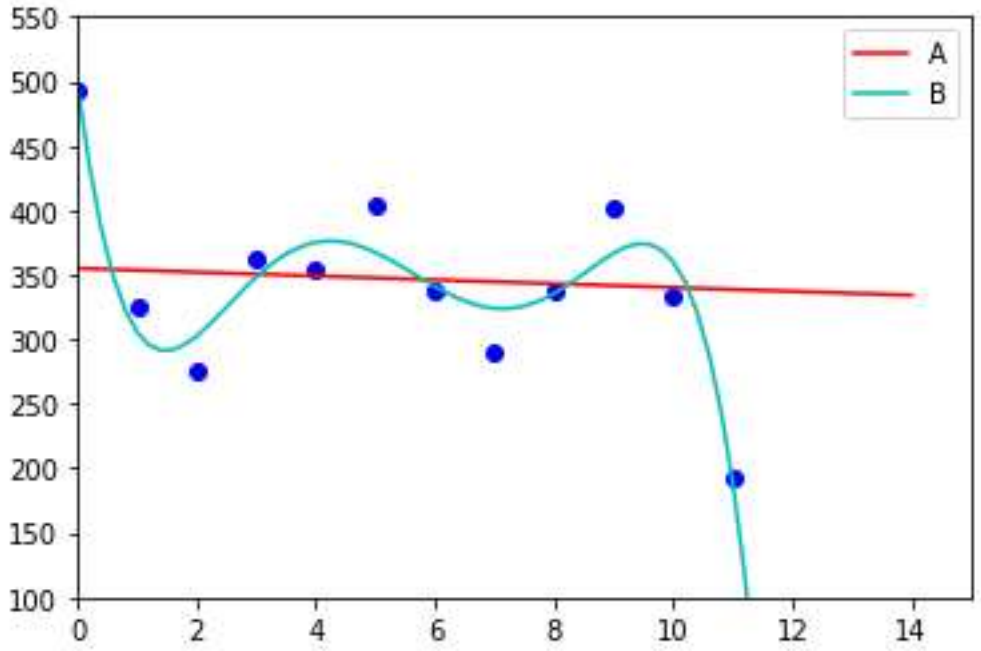
Overfitting: the model matches the training data **too well**



Remember: maximum accuracy on training data is not the goal

Training Screw-ups

Quarterly timber sales



A – linear trend B – curve fit

Q1	Q2	Q3	Q4	Q1
1093	1122	966	930	?

Model predictions for next Q1:

A	B
1007	0

Better fit for seen data, useless prediction for unseen data

Loss

We train by minimizing loss (of accuracy)

Revisiting Loss

Loss function:

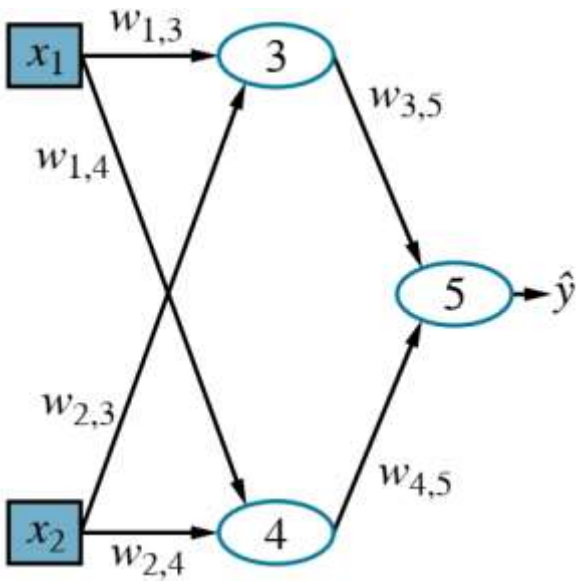
measure how much output differs from what we want

Output of this small network:

(depends on weights $\boldsymbol{w}$)

$\hat{y} = h_{\boldsymbol{w}}(x_1, x_2)$

Training sample: x_1, x_2, y

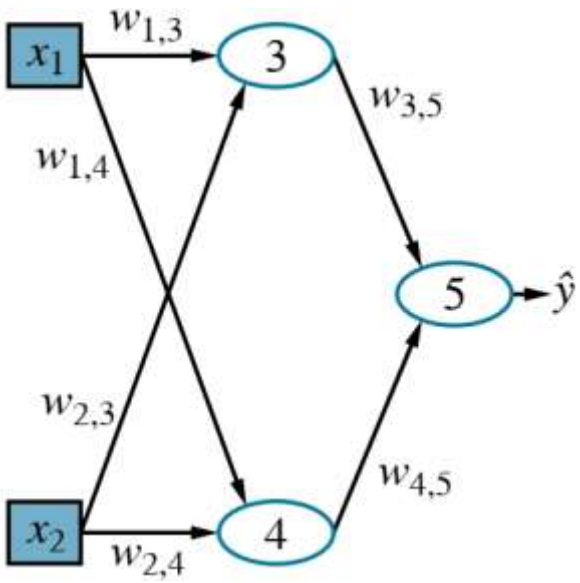


Neural Network is a Function

$$\hat{y} = h_{\boldsymbol{w}}(x_1, x_2)$$
$$= g_5(w_{0,5} + w_{3,5} g_3(w_{0,3} + w_{1,3} x_1 + w_{2,3} x_2)$$
$$+ g_4(w_{0,4} + w_{1,4} x_1 + w_{2,4} x_2))$$

One possible loss function:

squared loss
$Loss(\boldsymbol{w}) = (y - \hat{y})^2$



Log Loss

Loss can also be probabilistic:

The probability that output is y, given inputs x_1, x_2

$$P_{\boldsymbol{w}}(y|x_1, x_2)$$

Choose parameters $\boldsymbol{w}$ to maximise this.

We have multiple training examples (1), (2), …; this becomes

$$P_{\boldsymbol{w}}\left(y^{(1)}\middle|x_1^{(1)}, x_2^{(1)}\right) \times P_{\boldsymbol{w}}\left(y^{(2)}\middle|x_1^{(2)}, x_2^{(2)}\right) \times \dots$$

Log Loss

Easier to compute in log space (multiplication becomes addition):

minimize this function to maximise likelihood of data

$$Loss(\boldsymbol{w}) = -\sum_{j}^{N} logP_{\boldsymbol{w}}\left(y^{(j)} \middle| x_1^{(j)}, x_2^{(j)}\right)$$

N training examples

Output of network is the probability distribution $P_{\boldsymbol{w}}$

Use **softmax** as activation function of output layer

$$softmax(in_k) = \frac{e^{in_k}}{\sum_{k\prime}^{d} e^{in_{k\prime}}}$$

d – number of output neurons/units

in_k	$softmax(in_k)$	class
-0.12	0.107	cat
2	0.893	dog

Gradient

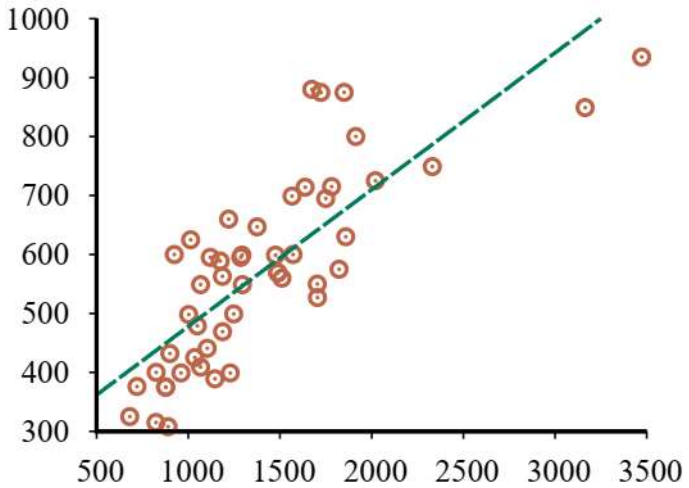
Gradient

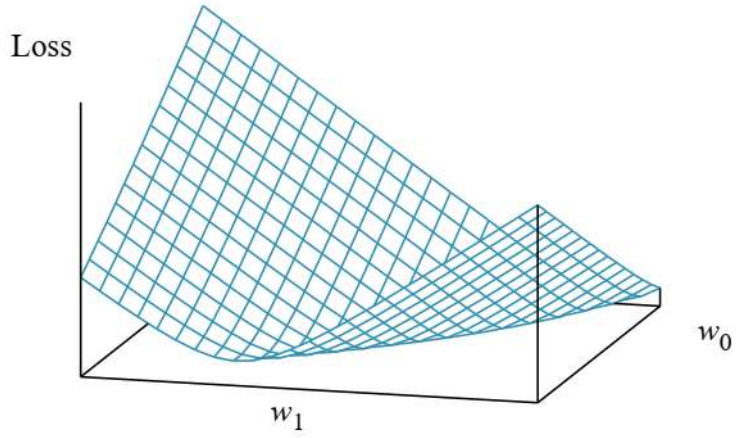
Gradient: for each point on loss surface,

which direction does the loss increase/decrease?

Linear model

$\hat{y} = w_0 + w_1 x$





https://youtu.be/Kbf6H5dwTFc (gradients in general)

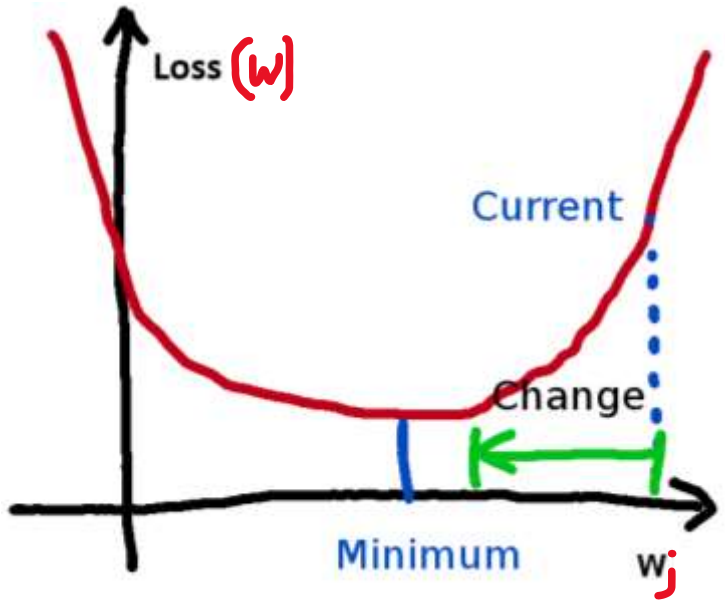
Gradient

Gradient defines a **vector** of growth for **each point** on the loss surface

Each weight is a dimension
gradient component for one dimension j

$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_j}$$

(partial derivative – think normal derivative;
pretend variables other than w_j are constant)



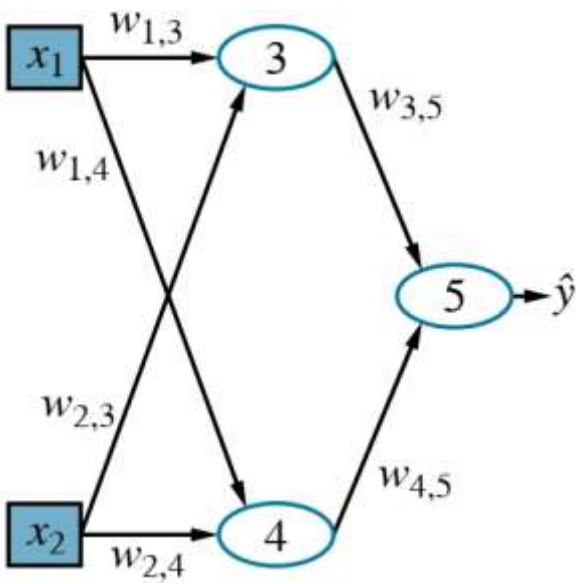
Gradient

For this network, using squared loss:

$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_{3,5}} = -2(y - \hat{y})g_5'(in_5)a_3$$

output of unit 3

$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_{1,3}} = -2(y - \hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)x_1$$



(derivations in AIMA 3rd ed. book 18.7.4)

Gradient

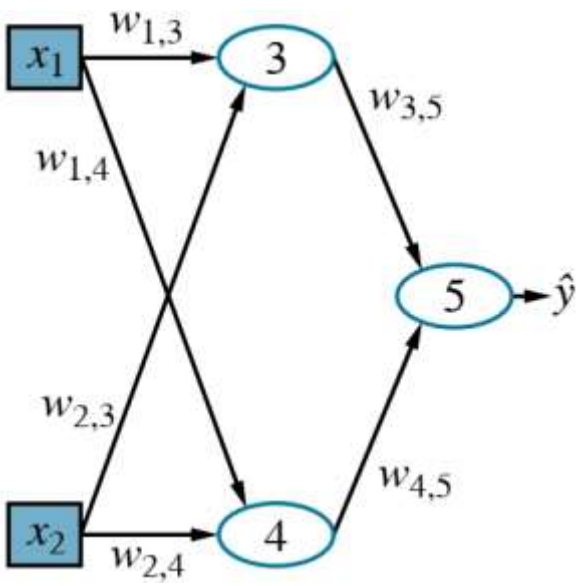
Modern NN implementations can do this **automatically**, backwards from outputs:

$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_{3,5}} = -2(y - \hat{y})g_5'(in_5)a_3$$

$$\Delta_5 = 2(y - \hat{y})g_5'(in_5)$$

comes from loss function

layer activation function



$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_{3,5}} = \Delta_5 a_3 \quad \text{(component of gradient } \nabla_{\mathbf{w}} Loss(\boldsymbol{w}) \text{ for } w_{3,5})$$

Gradient

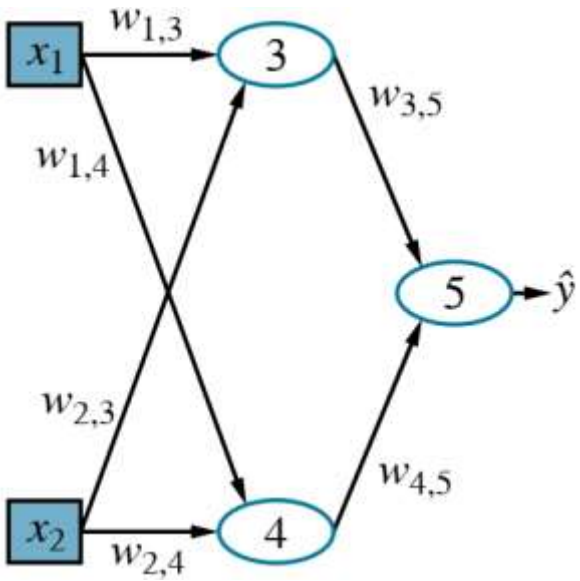
$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_{1,3}} = \overbrace{-2(y-\hat{y})g_5'(in_5)}^{\Delta_5} w_{3,5} g_3'(in_3) x_1$$

$$\Delta_3 = \Delta_5 w_{3,5} g_3'(in_3)$$

comes from layer type (connections)

layer activation function

$$\frac{\partial Loss(\boldsymbol{w})}{\partial w_{1,3}} = \Delta_3 x_1$$



g' -s: take derivative depending on layer type (softmax, logistic etc)

Batch

Weight update (each weight uses their respective gradient component)

$$\boldsymbol{w} \xleftarrow[\boldsymbol{update}]{} \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} Loss(\boldsymbol{w})$$

Different ways to compute loss:

- over all training examples
- over one random example (stochastic gradient descent)
- over m random examples (minibatch)

slow, sensitive to **learning rate** α

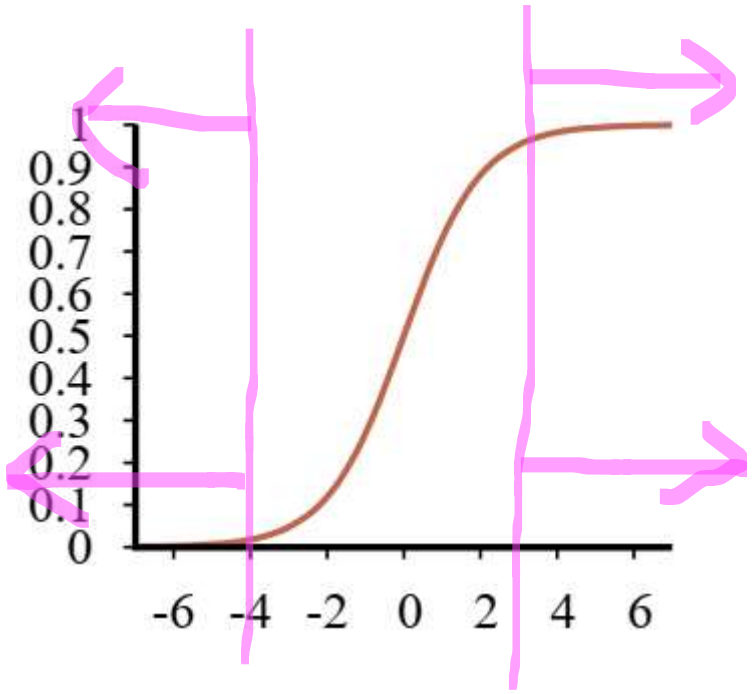
best balance in practice

Activation Functions

Logistic function

A.k.a “sigmoid”. Old school (until 2010)

- Differentiable
- Monotonous



Slow learning

(small gradient)

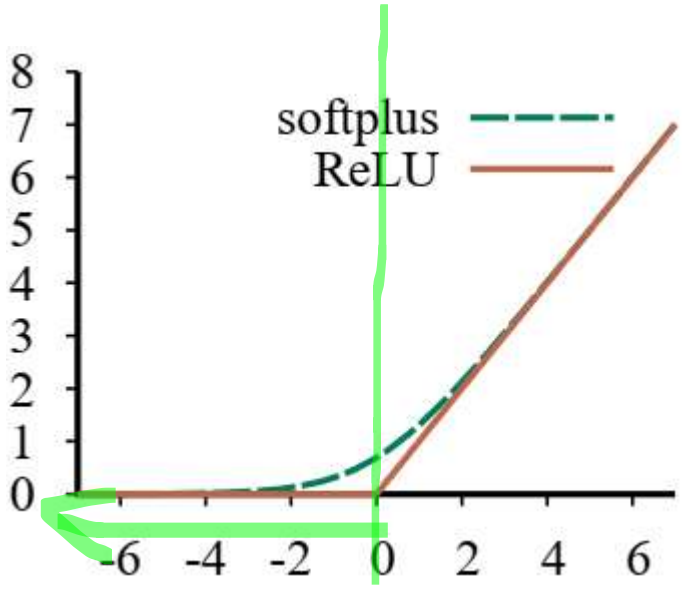
in marked directions

Rectified Linear Unit

ReLU:

Learns fast in

sloped region



Softplus:
smooth version
of ReLU

Zero gradient

in flat region

(not a problem unless it happens with all inputs)

Activations Based on Application

Softmax: for output layer in **classifiers** (uses inputs of entire layer)

$$softmax(in_k) = \frac{e^{in_k}}{\sum_{k\prime}^{d} e^{in_{k\prime}}}$$

Linear: output layer in **regression**

$$g(in_k) = in_k$$

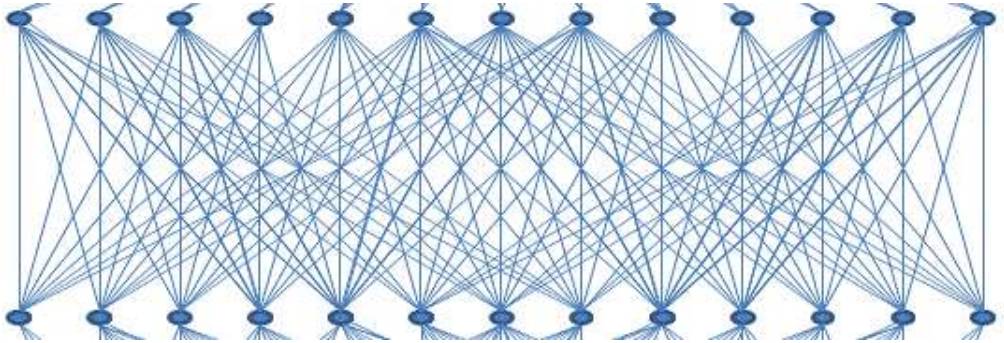
(when we want the network to predict a continuous value)

Structure

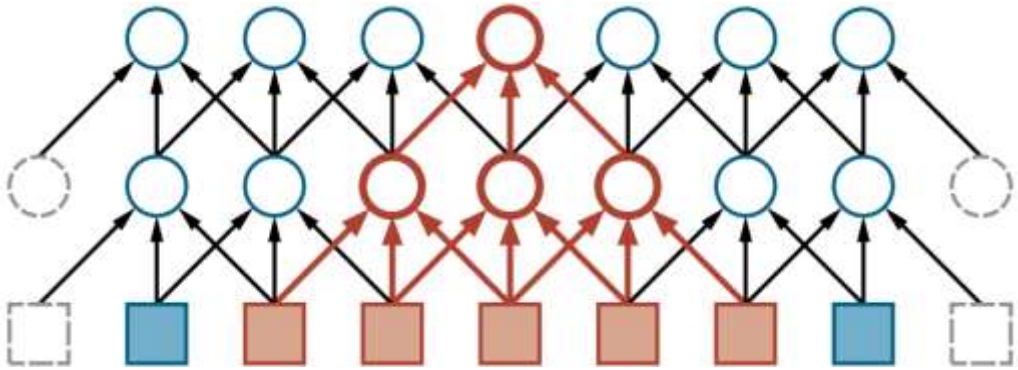
Fiddling with the connections to improve training efficiency

Convolutional Neural Networks

Fully connected layers:

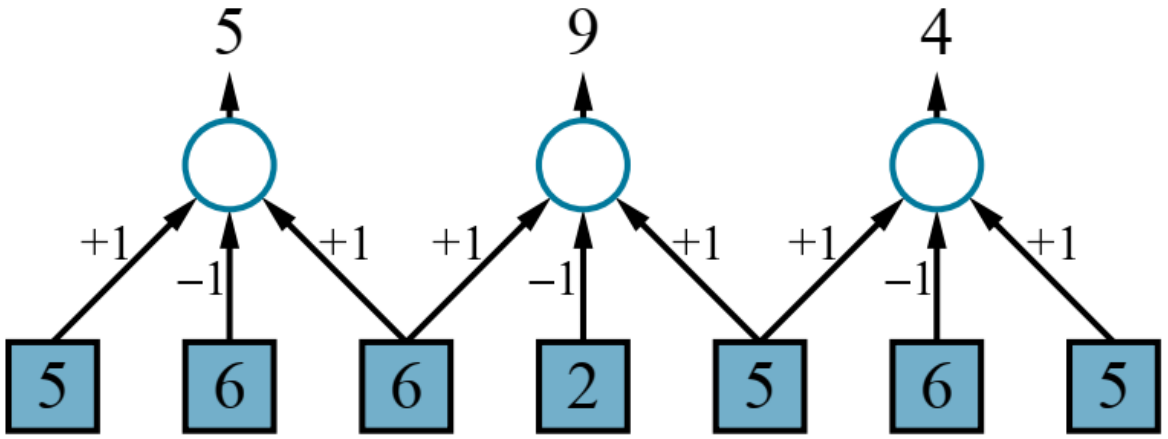


Convolutional layers (1D):



Convolutional Neural Networks

Convolution layer consists of filters or **kernels**:



[+1, -1, +1] are learned weights and **shared**/repeated over input units

This filter detects a lower (darker) value “2” in the middle

Filters or Kernels

Convolutional layer has multiple kernels: each learns a pattern

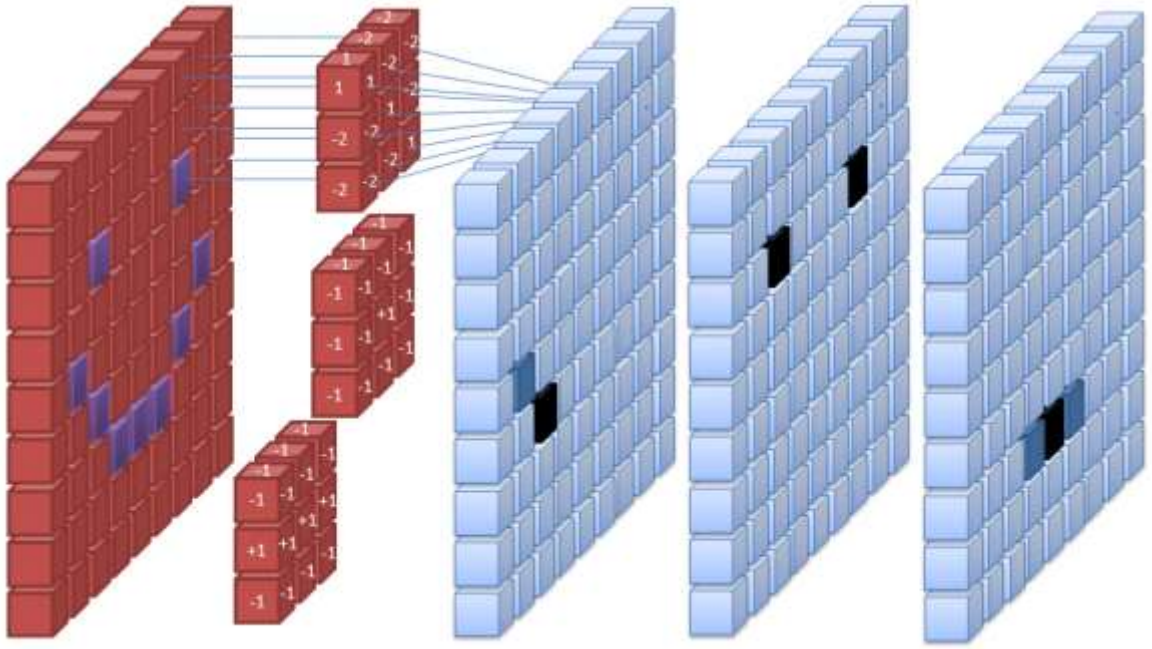


Image: Wikimedia Commons

Efficiency in CNNs

A fully connected network can learn the same task

(e.g. 0 weights for connections not present in convolutional layers)

CNN will learn faster: less weights

Inputs and activation of 8 5x5x1 convolution filters, MNIST data:

CNN in Action

Andrej Karpathy's browser CNN demo:

https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html